(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
6 June 2002 (06.06.2002)

PCT

(10) International Publication Number
WO 02/44835 A2

(51) International Patent Classification[7]: G06F

(21) International Application Number: PCT/US01/43154

(22) International Filing Date:
26 November 2001 (26.11.2001)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
09/722,321 28 November 2000 (28.11.2000) US

(71) Applicant and
(72) Inventor: GINGERICH, Gregory, L. [US/US]; 12939 Wood Crescent Circle, Herndon, VA 20171 (US).

(74) Agents: TURNER, Richard, C. et al.; Sughrue Mion, PLLC, 2100 Pennsylvania Ave., NW, Suite 800, Washington, DC 20037-3213 (US).

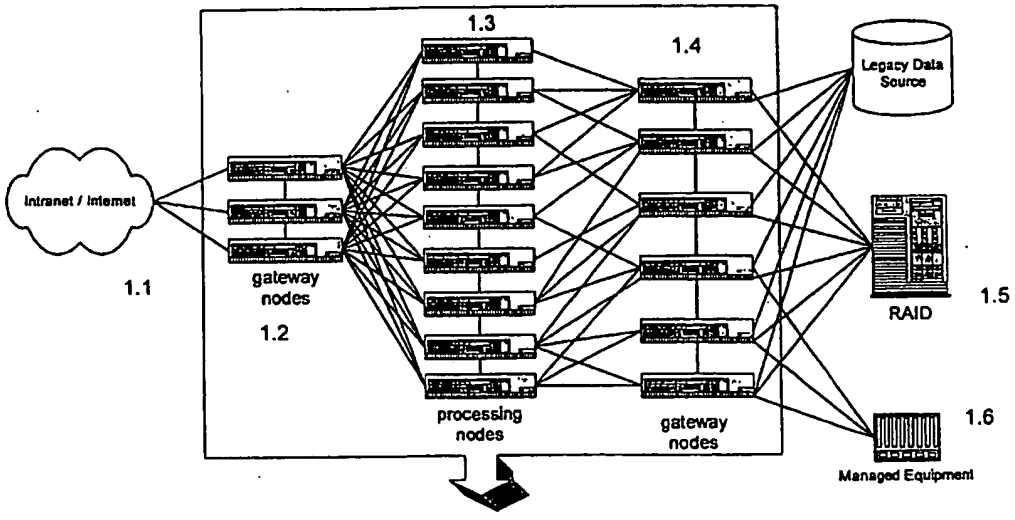
(81) Designated States *(national)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, OM, PH, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, TZ, UA, UG, US, UZ, VN, YU, ZA, ZM, ZW.

(84) Designated States *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:
— *without international search report and to be republished upon receipt of that report*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

(54) Title: A METHOD AND SYSTEM FOR SOFTWARE AND HARDWARE MULTIPLICITY



(57) Abstract: Methods for creating a scalable, fault tolerant computing platform on a network, said platform comprising a plurality of nodes are disclosed. The method comprises initiating melding during initialization, wherein during said melding said plurality of nodes are treated as a collection and each node from said plurality of nodes joins said collection, and during melding at least a first object router is assigned; migrating objects between said plurality of nodes; and replicating objects across said plurality of nodes. Systems similar to the disclosed methods are alsodisclosed. Computer program products implementing the methods are also disclosed.

WO 02/44835 A2

A method and system for software and hardware multiplicity

I. DESCRIPTION

I.A. Field

This disclosure generally teaches a method and system for deploying and executing software objects in a network of processors. The disclosed techniques are embodied in computer platform systems, methods for creating a computer platform and related computer program products.

I.B. Background

Conventional large-scale software systems often exceed the limits of the computational power that a single processor or machine can provide. It is therefore desirable to create software systems capable of running across many processors or machines. For the most part, the emphasis in conventional systems has revolved around hardware and operating system level solutions. But with widespread use and availability of the Internet and ubiquity of networking, the abstraction level has moved up. This has led to research and development of systems based on remote objects.

While implementations of remote objects have varied, the general approach involves, for example; marshalling objects, method invocations, and data attributes into a byte stream for transit across a network. The byte stream representing the remote object is then demarshalled by another machine and executed 'remotely.' Common implementations of such remote object systems include: DCE, CORBA, XML, COM/DCOM, DOM, SOAP, and RMI. These technologies enable machines on a

network to execute code remotely located on another machine. This in turn opens possibilities for the creation of scalable systems.

In practice, the number of machines (and therefore the scalability) of such systems is often limited to less than a dozen. This is primarily due to the interdependencies between layers of software, "synchronization" of stubs and skeletons, and the need to spread the system load efficiently across these machines. This is further complicated by the difficulty of properly configuring a large number of systems and deploying the correct versions of software across all machines. Conventional systems using this abstraction layer (remote objects) additionally suffer from the lack of open, flexible, and unified automation controlling resources, objects and redundancy within the system.

The end result is that most conventional remote object systems in place today suffer from outages caused by improper configuration or deployment of software and hardware. Additional outages are incurred (and considered normal) during the upgrade or replacement of software and hardware elements. These upgrades are often performed at 'off' hours (evenings/weekends) when system availability is considered less critical. With the advent of ubiquitous computing and the growing importance of the Internet, however; such outages are no longer tolerated as customers and vendors now expect true 24 x 7 system availability.

II. SUMMARY

To overcome the above problems in conventional technologies, an object of the present teaching is to provide a solution that enables systems to remain highly

available at all times and eliminates the need to disable system access during upgrades and maintenance.

To meet the objectives of the disclosed teaching, a method for creating a scalable, fault tolerant computing platform on a network, said platform comprising a plurality of nodes, said method comprising initiating melding during initialization, wherein during said melding said plurality of nodes are treated as a collection and each node from said plurality of nodes joins said collection, and during melding at least a first object router is assigned; migrating objects between said plurality of nodes; and replicating objects across said plurality of nodes.

Preferably, during melding each node joins the collection using a process comprising: the node finding a nearby object router if the object router can be found, otherwise becoming the object router; and the node reporting available resources to the object router.

Preferably, during melding each node joins the collection using a process further comprising assigning network address for the node; and updating components of the node.

Preferably, the available resources comprise memory, and at least one processor.

Still preferably, the available resource comprises network bandwidth.

Still preferably, the available resources comprise drive storage space.

Still preferably, said updating is done using data stored in a code version server.

Still preferably, it is ensured that at least one mirror of an object router exists all the time.

Preferably, the object router manages resources and distributes load evenly across the plurality of nodes.

Preferably, the object router maintains knowledge of locations of at least a subset of all nodes in the computing platform.

Preferably, the object router relocates or replicates objects within the plurality of nodes.

Preferably, said migration of an object is performed using a process comprising locking the object; converting the object to a byte stream at a first node in the plurality of nodes; transferring the byte stream across the network; converting the byte stream back to the object at a second node in the plurality of nodes; updating a location of the object in the object router; and unlocking the object.

Preferably, said replication of an object is performed by a process comprising: locking the object at a first node; forming a duplicate object ; locating the duplicate object at a second node; and unlocking the object.

Still preferably, the method further comprises subscribing to events associated with the object.

Still preferably, said duplication is performed by creating a byte stream from the object and converting said byte stream into a duplicate object.

Preferably, the computing platform allows for evolution, wherein said evolution provides for transitioning of system software and hardware such that a seamless upgrade is possible.

Another aspect of the disclosed teachings is a method for performing evolution, said evolution being performed to enable a seamless transitioning of

software, said method comprising: evaluating a software or hardware to be installed or upgraded; creating a dependency hierarchy tree of all objects in the computing platform at compile time; performing object class diff comparison between old and new versions of at least one class of objects; creating a transform function for each state variable that is changed; selecting an object that is an instance of the selected object class; locking the selected object; creating uninitialized instance of the new version of the selected object; initializing variables of the new object class instance by applying appropriate transform functions to the old object version; unlocking the selected object; and repeating relevant steps for each instance of the object class.

Still preferably, the method comprises repeating relevant steps for each object class in the computing platform.

Still preferably, the method comprises deploying client-side software.

Still preferably, the method comprises destroying older version of the objects if multiple versions of the objects are not required.

Preferably, the object router maintains multiple instances of each of said objects.

Preferably, a failure of a first node from the plurality of nodes is detected when a proxy object from a second node is unable to reach the first node.

Preferably, a failure of a node from the plurality of nodes is detected when the node fails to respond to heartbeat requests from the object router.

Preferably, a failure of node from the plurality of nodes is detected when the node fails to report either status or statistics to the object router.

Preferably, when a node from the plurality of nodes fails, said node being called a failed node, said at least one object router responds using a process comprising: notifying other object routers other than said at least one object router; selecting an object that resided in the failed node; making replicated object corresponding to the selected object residing in nodes other than the failed node primary; locating additional computing resources within the computing platform; creating new replicates for the newly created primary objects and locating them in nodes other than nodes in which they originally resided; and notifying said other object routers of new location of the new replicates.

Still preferably, the method comprises notifying a system administrator about the failed node being off-line.

Still preferably, the method comprises repeating relevant for all objects in the failed node.

Preferably, when an object router from said at least one object router fails, said object router that fails being called a failed object router, the computing platform responds using a process comprising: utilizing services of object routers other than said failed object router; replicating the failed object router to create a replicate object router; locating the replicate object router on at least one functional node ; and notifying at least a subset of the nodes of a new location of the replicate object router.

Still preferably, the method comprises notifying at least a subset of other object routers of the new location of the replicate object router.

Preferably, said object router is aware of at least a subset of other object routers.

Another aspect of the disclosed teachings is a network computing platform comprising a plurality of nodes, said system comprising: at least an object router, wherein said object router is capable of managing resources and distributing load evenly across the plurality of nodes, and wherein said object router maintains a knowledge of locations of at least a subset of all nodes in the computing platform.

Preferably, the object router relocates objects within the plurality of nodes.

Preferably, the object router maintains multiple instances of each object that is deployed in the platform.

Preferably, each of said nodes comprise: an object melder that treats the plurality of nodes as a collection that performs melding to create said plurality of nodes and said at least one object router.

Preferably, the platform further comprises an object versioner that maintains multiple versions of all objects within the platforms.

Preferably, the platform further comprises an object replicator that replicates an object to maintain a duplicate of the object in a node different from the node in which the object resided.

Preferably, the platform further comprises an object migrator that enables an object to migrate from one node to another.

Preferably, the platform further comprises a resource monitor.

Preferably, the platform further comprises a resource controller.

Preferably, the platform further comprises an application object.

Preferably, each of said nodes further comprise: a class loader that is responsible for loading classes of objects, wherein the class loader permits multiple versions of a same class to co-exist.

Preferably the platform further comprises an evolution module, wherein the evolution module allows the computing platform to perform evolution, wherein said evolution provides for transitioning of system software and hardware such that a seamless upgrade is possible.

Preferably, a failure of a first node from the plurality of nodes is detected by the platform when a proxy object from a second node is unable to reach the first node.

Preferably, a failure of a node from the plurality of nodes is detected by the platform when the node fails to respond to heartbeat requests from the object router.

Preferably, a failure of node from the plurality of nodes is detected by the platform when the node fails to report status or statistics to the object router.

Preferably, when a node from the plurality of nodes fails, said node being called a failed node, said at least one object router is capable of notifying other object routers other than said at least one object router.

Preferably, when a node from the plurality of nodes fails, said node being called a failed node, said at least one object router is capable of notifying other nodes

Preferably, when a node from the plurality of nodes fails, said node being called a failed node, said at least one object router is capable of making replicated object corresponding to at least a subset of objects residing in nodes other than the failed node primary.

Preferably, when a node from the plurality of nodes fails, said node being called a failed node, said at least one object router is capable of locating additional

computing resources within the computing platform and creating new replicates for newly created primary objects and locating them in nodes other than nodes in which they originally resided.

Still preferably, the platform is capable of notifying said other object routers of new location of the new replicates.

Still preferably, the platform is capable of notifying a system administrator about the failed node being off-line.

Preferably, when an object router from said at least one object router fails, the computing platform is capable of utilizing services of object routers other than said failed object router.

Preferably, when an object router from said at least one object router fails, the computing platform is capable of replicating the failed object router to create a replicate object router and locating the replicate object router on at least one functional node.

Still preferably, the platform is further capable of notifying at least a subset of the nodes of a new location of the replicate object router.

Preferably, said object router is aware of at least a subset of other object routers.

Another aspect of the present invention is an object router for a network computing platform comprising a plurality of nodes, wherein said object router is capable of managing resources and distributing load evenly across the plurality of nodes, and wherein said object router maintain a knowledge of locations of at least a subset of all nodes in the computing platform.

Preferably, the object router relocates objects within the plurality of nodes.

Preferably, the object router maintains multiple instances of each object that is deployed in the platform.

Preferably, when a node from the plurality of nodes fails, the object router is capable of notifying other object routers other than said object router.

Preferably, when a node from the plurality of nodes fails, the object router is capable of making replicated object corresponding to at least a subset of objects residing in nodes other than the failed node primary.

Preferably, when a node from the plurality of nodes fails, the object router is capable of locating additional computing resources within the computing platform and creating new replicates for the new primary objects and locating them in nodes other than nodes in which they originally resided.

Preferably, the object router is aware of at least a subset of other object routers.

The disclosed teachings also include a computer program product including a computer-readable medium comprising instructions to enable a computer to implement the disclosed methods.

III. BRIEF DESCRIPTION OF THE DRAWINGS

The above objectives and advantages of the present invention will become more apparent by describing in detail preferred embodiments thereof with reference to the attached drawings in which:

FIG.1 (a) shows a general network platform to which the techniques disclosed can be implemented.

FIG.1(b) shows a platform embodying the disclosed techniques.

FIG.2 shows the architecture of a node with the disclosed platform.

FIG.3 (a)-(c) show three examples of replication.

FIG.4 shows an object version tree.

FIG.5 (a)-(b) show an embodiment of embedded object versioning.

FIG.6 shows an embodiment of object transformation.

FIG.7 shows an example of classloader chaining.

FIG.8 shows an example of classloader hierarchy with multiple object versions loaded.

FIG.9 shows examples of locations where a system embodying the disclosed techniques can be installed.

IV. DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

Preferred Embodiments of the disclosed teachings and techniques, are provided that enable the efficient distribution and 'routing' of software objects across many machines while automating the configuration and upgrade of both hardware and software elements.

The preferred embodiment described herein creates a highly scalable, fault tolerant computing platform that is as easily maintainable as a Redundant Array of Inexpensive Disks (RAID) based drive array. It could be referred to as a RAIC™ "Redundant Array of Inexpensive Computers™".

Using the disclosed techniques, the preferred embodiment incorporates characteristics that allow for dynamic hardware and software reconfiguration and growth. It achieves this by introducing several unique features into the platform as follows:

- Melding: Machine/Node configuration is highly automated.
- Auto Install: Software is automatically loaded and distributed across the nodes.
- Object Routing: Objects are automatically routed across the nodes to increase scalability and reliability.
- Automated Recovery: The computing platform is self-healing in nature, when node or component failure occurs.
- Evolution: Through automated management of software versioning and dependencies, the computing platform can be seamlessly upgraded without any noticeable interruption of services to end-users.

Operation of the preferred embodiment may be subdivided into a number of composite processes. It should be noted that these composite processes or process steps may be used separately or in combination, as claimed. The scope of the disclosed techniques is not limited to any particular combination.

A typical network computing platform is shown in FIG.1. A first plurality of gateway nodes 1.2 are connected to the Internet 1.1. A plurality of processing nodes 1.3 are connected to the first plurality of gateway nodes 1.2. A second plurality of gateway nodes 1.4 are connected to the processing nodes 1.3. Legacy Data Sources 1.4, RAID 1.5 and management equipment 1.6 are connected to the second plurality of gateway nodes. FIG.1(b) shows a platform embodying the disclosed techniques. The object routers are marked OR. The objects are marked OJ and replicators are marked "r".

FIG.2 shows a node architecture for the preferred embodiment. During system initialization/boot performed at 2.1, a signaling process called melding

occurs. The melding process is performed by a melder. During the process of melding, each node discovers other nodes and components present in the network and informs the network of its system resources including processing power, memory, storage space, and network capabilities. Melding also includes the assignment of an object router 2.2 responsible for directing object access requests to and from each node. Object routers monitor resource utilization across all nodes and distribute requests accordingly. Object routers also maintain multiple instances of each object (located on disparate nodes) to prevent single points of failure within the network. It should be noted that in the disclosed system, object routers may serve as a replacement for more primitive remote object lookup services (including but not limited to services such as COSNaming specified by OMG).

In order to handle the shutdown or failure of an individual node within the network, processes called migration and replication, performed at 2.3, provide fault tolerance and the ability to move live (executing) objects across nodes. Migration is performed by the object migrator. Replication is performed by the object replicator. These processes are also utilized during hardware upgrades, during which time individual nodes are effectively shutdown and unavailable. While individual nodes are unavailable, this remains transparent at the system level and the overall system of nodes remains highly available to all users.

In order to accommodate software changes, a process named evolution utilizes the above named capabilities in conjunction with transform functions and sophisticated object versioning techniques. This enables seamless transitioning of software components while keeping the system available to end users.

Some of the processes and components of the platform described briefly above are discussed in further detail herein below.

IV.A. Melding

Melding is the process of individual computing nodes joining the larger 'collective' of nodes.

During melding, a method such as the internet protocol DHCP is utilized to assign IP or other network addresses for each node.

Then the bootstrap of core components including a module/class loader occurs. It should be noted that, each node's core components are updatable from data stored in a "code version server" (http, ftp or other network protocol server enabling data exchange) during startup.

Next, the location of an object router is determined. After the node boots, it attempts to find a nearby object router node utilizing multicast packets, broadcast packets, or a "well known" (preconfigured) lookup address (such as in an internet protocol DNS server). If no object router is located, the node loads the 'router module' and becomes an object router node. In the preferred embodiment, it is ensured that at least one router 'mirror' exists at all times.

Then, resource reporting is performed. During this reporting, the node reports its available hardware and software resources to its object router . Statistics reported can include available memory, processing units, drive storage space, network bandwidth, software installed, and software versions.

IV.B. Object Router

The object router keeps track of available resources. The object router tracks utilization of resources within the nodes it manages. It is responsible for distributing load evenly across its nodes.

The object router also performs remote object tracking. The object router maintains a knowledge of location of objects located within its nodes. It is responsible for relocating objects across nodes when required. It should be noted that location can include but is not limited to a network address and port number, DNS name, URL, or remote object reference as specified in a remote object system.

IV.C. Resource Monitoring/Control

The preferred embodiment provides for resource monitoring and control. Resource monitoring is performed by a resource monitor module. Resource control is performed by the node resource controller. Both of these modules are utilized by the object router to monitor and control the state of individual nodes.

In a Java based implementation, these modules can be implemented utilizing JNI. In such a case, standardized interfaces for accessing operating system kernel structures such as process IDs, threads, memory utilization, network activity, disk activity, and related statistics, are created. These interfaces are then implemented with custom JNI wrapper objects.

In such a Java based implementation, JNI wrapper objects also provide a mechanism to control process priorities, threads, operating system configuration,

and system reboot. Such wrapper objects unify these operations across a variety of operating systems.

IV.D. Migration:

Migration is the process of moving an object from one node to another. Migration is performed by the object migrator.

First an object is locked. This is done to prevent state inconsistency during its migration.

Then the object is serialized. During this step, the object is converted to a byte stream for transit across the network.

Then the object is relocated. During this step the byte stream is converted back to an object at an available node.

Finally the object is unlocked. The location of the object is updated in the object router and made available for use.

IV.E. Replication

Replication is the process of duplicating an object across multiple nodes. Replication is performed by the object Replicator. FIGs.3(a)-(c) show three embodiments of object replication.

IV.E.1.Mode 1

FIG.3(a) illustrates one example called "mode 1" of replication whereby an operation is initiated by the client of an object. This client request may come from either objects located on another node or server, or a software application. The operation (or method call) is sent to a local proxy object in the originating node,

which (after having obtained location of the remote, replicated objects from an object router), sends that request to the first (or primary) remote object located at Node A. Node A immediately forwards the same marshaled operation to a replicated object at Node B, then the primary object at Node 'A' proceeds to perform the operation, modifying its state as required. The replicated object at Node B simultaneously performs the same operation, and sends any result set (return parameters) back to the proxy object at the node of origination. Optionally, a checksum value may be calculated at both remote nodes on either the result set, the internal state variables of the object, or both. This checksum step is utilized to detect errors, and correct faults that occur.

The mode of replication illustrated in FIG.3(a) has the advantage that it minimizes network traffic between the nodes and reduces the likelihood of network congestion at the originating node. It also "guarantees" (to a degree) that all replicates are kept synchronized with each operation. A potential negative of this approach is increased message latency due to the additional network 'hop' required from Node A to Node B.

Note that this approach can easily be extended by adding additional nodes in the chain between Node A and Node B with slight latency penalty for each additional node/replicate added.

IV.E.2.Mode 2

FIG.3(b) illustrates a second example, called "mode 2", of replication where an operation is initiated by the client of an object (which may come from either objects located on another node or server or a software application). The operation (or method call) is sent to a local proxy object in the originating node, which (after

having obtained location of the remote, replicated objects from an object router), sends that request simultaneously to both remote objects located at Nodes A and B. The replicated objects at Nodes A and B perform the operation in parallel, and the 'primary' remote object is responsible to send a response to the proxy object located in the originating node.

Optionally, checksum values of result sets, internal state variables, or both may be sent to the local proxy object, or as a "cross-check" between the remote objects. This checksum value is utilized to detect errors and insure consistency.

Note that this approach can easily be extended by adding additional nodes that the proxy communicates with, however a tradeoff in terms of network bandwidth utilization is to be expected. This approach reduces expected message latency, but increases overall network utilization.

IV.E.3.Mode 3:

A third technique, shown in FIG.3(c), involves the proxy object communicating directly with a primary remote object, whose state changes are mirrored to one or more secondary remote object(s) that have 'subscribed' to events reflecting state changes within the primary object. Again, checksum messages can be added to improve system robustness.

All three techniques, modes 1-3, utilize a local proxy object in order to isolate the application programmer from the complexities of the replication process. The proxy object serves as a local surrogate for the remote objects, thereby eliminating the need for an application programmer to write network, checksum, and error recovery code that can be handled by the proxy and object router working together.

In practice, one or more of these techniques may be utilized to maintain replicated state between remote objects, and hybrid approaches may also be feasible.

In all the modes, the following operations are common:

Object locked: object is locked to prevent state inconsistency.

Object cloned: the object is duplicated through serialization.

Object relocated: the duplicate object is remotely located at a secondary node.

Proxy object / Event subscription: object operations are replicated.

Object unlocked: object is unlocked and made available for use.

IV.F. Evolution:

Evolution is one of the more unique and complex processes within the preferred embodiment platform. It relies on a combination of other features, including object migration, advanced versioning, specialized classloaders, and transform functions.

IV.F.1. Compiling/ pre-post processor:

This refers to processes performed either prior/during/or post-compilation. Additional steps are performed on either the source or object code to insert the additional versioning information required by evolution (prior to software deployment). This is a step that would normally be executed by the software developer or a software system integrator or vendor. Several approaches are described below.

a) Pre-Compilation

Prior to executing a compiler on the source code, a preprocessor can be executed that parses the source code and inserts additional (source) code within each class that stores information reflecting the version required for each external class utilized by that class. For instance, class 'A' relies on classes 'B', 'C', and 'D.' Prior to compilation, code would be inserted into class 'A' indicating the versions of classes 'B', 'C', and 'D' that are required, along with the version (most likely a computed hash of the class definition) of class 'A'. This type of a process is sometimes referred to as "source code insertion"

b) During Compilation

A specialized source code compiler can be constructed that inserts the versioning information, similar to that shown in FIG.5(a), automatically during the compilation process. For each object being compiled, the compiler's "external symbol table" contains references to each external class or method referred to by the class being compiled. An extended compiler design would add version information to the external symbol table and store this information in the produced object code for each class. This would be an extension or revision to the language's compiler.

c) Post-Compilation:

It is possible to read/process Java object bytecodes (or other generated object code) after compilation and insert bytecodes which contain the versioning information. This is performed typically as shown in 5(a), the "embedded object versioning" diagram. A software tool to perform this process would typically be executed by a software developer or independent software vendor (ISV). The biggest single advantage of this approach is that it would be feasible to apply this to third party software for which source code is unavailable. Methods 1 and 2 above do not work as well for software without available source code. This could be referred to as "object code insertion."

It should be noted that Java Object ByteCodes are roughly equivalent to assembly/machine code produced by the compilation of C/C++ software. Similar methodologies may apply for non-Java based systems.

An alternate approach to storing the versioning information directly within the object classes (which currently *appears* to be the favored approach) is to store this information externally as a lookup table stored in the node(s) or made available to the node(s) on some alternate data server or node. Each node's Classloader could then access this information during class loading without it being directly part of the object.

For all scenarios:

The specialized ClassLoader residing within each node relies on the 'extended versioning' information that is made available for each class to load the appropriate

version of each object required in the dependency tree. When the Node's Classloader loads Version '1' of Class 'A' it gains the knowledge that it needs to acquire/load version '2' of Class 'B' and version '3' of Class 'C.'

Also, while Non-Java based classloaders are not commonplace today, in theory such constructs could be created (i.e., a C++ Classloader or an Oracle™ database Classloader) for any object based language. The version numbers of the classes being compiled would be stored along with the version numbers of the external classes.

Utilizing computed hash values for each class is a preferred method for producing version numbers (due to uniqueness), although arbitrary version numbers [such as labels from a configuration management products like ClearCase, SourceSafe, CVS, or Continuous could also be utilized]. This process may be used in conjunction with source code repository/configuration management tools – such as the above mentioned products.

IV.F.2. Version hierarchy:

FIG.5(a) shows an example of version information that may be associated with objects/classes. This is intended to illustrate the relationship between object class versions. Version numbers illustrated in this figure are arbitrary. Every object class is assigned a "VersionID," however the version table may also be stored externally to the class.

The version hierarchy is formed during either:

a) The execution of the source code pre-processor

b) The execution of the "modified" compiler

c) The execution of the object code post-processor.

The version hierarchy is normally 'implicitly stored' at runtime/deployment time within the objects:

Ie: Object Class 'A' has knowledge that it depends on 'B' and 'C'

Object Class 'B' knows that it relies on Class 'D'

Object Class 'C' knows that it relies on Class 'E'

This would implicitly form the hierarchy of:

A -> B -> D

A -> C -> E

Or, pictured another way:

A

B C

D E

This tree could expand indefinitely – and the 'terminal' nodes of the tree are (ideally) the Operating System and/or Hardware component identifiers.
(In the absence of available information through the hardware layer, the deployment process will take a 'best effort' approach to deployment and attempt to find a close fit.)

For example, Object Class 'C' may depend on Object Class 'E' which depends on Java JRE 1.2 which depends on Solaris 8, OS patch version 482 which depends on 128MB RAM, 200MB drive storage, and a 100MHZ SPARC chip version 'E' or later.

Within traditional systems, object dependencies are not normally utilized outside of the compiler and normally do not extend beyond the confines of the language. After compilation, such information is typically discarded. The disclosed system stores and utilizes both object dependencies and versioning information throughout deployment and runtime of the objects. This enables enhanced software object tracking and transitioning. FIG.4 shows an object class version tree. In this figure, "v" represents the version number. The Object Class names in the figure are arbitrary and this diagram can also illustrate computer language versions, operating system versions, and hardware versions. (ie: "Altitude" could be replaced with "Java" or "Linux" or "Intel Pentium").To form the Version Tree Hierarchy the following information sources are relied upon:

a) Compiler (or compiler-like) algorithms that ensure method-signature compatibility among objects and components to be deployed.

b) Programmer specified or Configuration Management Tool specified version dependencies (CVS, Continuus, Clearcase, SourceSafe, etc).

c) Information specified by vendors of third party software:
IE: Oracle, Sun, IBM, and other vendors specify the platform on which their software may be deployed and often provide this information in the form of "Release Notes" documentation.

d) Information system producers (or other outside parties) gather through other means and add to a repository of compatible configurations.

IV.F.3. Version lookup table

FIG.5(b) provides an illustration of how the version table may be stored separately from the object class. This information may be stored in one or more nodes or in an external server or database. A lookup table exists within each object, consisting of a list of all Objects required by this object, and the appropriate version of each object to load. The 'version lookup table' should be apparent from the information provided in the hierarchy and pre-processor sub-sections. The lookup table is a set of relationships between specified Classes and the versions of the Classes they depend on. A lookup table exists within each object, consisting of a list of all objects required by this object, and the appropriate version of each object required. The lookup table is a set of relationships between specified Classes and the versions of the Classes they depend on. This is stored either:

a) within each object to be deployed.

b) within one or more node(s).

These relationships are utilized by the object Classloader in order to determine dependencies when objects are loaded within the node(s).

IV.F.4. Custom Classloaders

Specialized classloaders are utilized that permit simultaneous loading of multiple object versions. Traditional systems (and their classloaders) only permit the loading of a single version of an object class at any given time.

IV.F.5. Transform f(x)/process:

FIG.6 illustrates an object/class transform mapping scenario for an object of class "Employee". It should be noted that the transform functions could be

combined into a single transform function, or further subdivided into more functions. In this figure, V1 is version 1 and V2 is version 2.

As shown in FIG.6, for each object instance the following steps are performed:

1) Lock Object

2) Create version 2 (v2) proxy/instance

3) Apply transform function

4) Subscribe Change Events

5) Unlock Object

For each new version of an object, a series of transform functions are created. The purpose of the transform function is to 'translate' an object from a prior version to the current one. For many common transforms, pre-defined functions may exist. For more difficult transforms, programmer intervention may be required.

IV.F.6. Preferred Embodiment of Evolution

The preferred embodiment of the general evolution process is described herein.

1) Evaluate the software to be installed or upgraded:
   a. Create a version hierarchy tree of all objects at compile time.
   b. For each class of objects, perform an object class 'diff' comparison between the old class versions and newly created versions.
   c. For each state variable or method signature that has changed, create a transform function.
   d. At deployment time: for each class of objects to be upgraded/installed, do:

i. Loop for all object instances of each class:

1. Lock object to prevent state changes
2. Create an uninitialized instance of the new object class version and corresponding proxy objects and replicates as required.
3. Initialize the variables of the new object class instance by applying the transform function to the prior object version.
4. Subscribe to change events.
5. Unlock object.

e. Upon completion, there will be a 'mirror' copy of the original system, containing new versions of all objects. At this point, new "client-side" software is deployed.

f. Upon completion of new "client-side" software, prior versions of objects may be disposed of. Alternately, multiple versions may be retained, but with significant cost of resource utilization.

The preferred embodiment incorporates several extensions to objects within the platform to facilitate the mechanisms required for migration and replication. Objects within the preferred embodiment described herein support the following 'base class' extensions:

- **Object locking:** Objects support distributed read and write locking mechanisms.

- **Change Events:** Change events are generated when a state change occurs within the object. These change events may be remotely subscribed to across nodes.
- **Versioning of objects:** Objects are assigned a version ID that is either the computed hash of their interface or an assigned value by either the developer or configuration management tool. Additionally, objects are aware of the version IDs of objects they utilize.
- **Objects serializable:** Remotely accessed objects have the ability to be converted to a byte stream for transit across a network.
- **Object Extension:** Objects are run-time extendible through object references.

IV.F.7. Evolution of Hardware

Evolution of Hardware is handled in a somewhat different (though related) fashion than Software:

During upgrade or maintenance of a piece of hardware:

a) The 'state' of the node to be upgraded, replaced, or maintained is changed to reflect a 'down', 'offline,' or 'upgrading' state within the ObjectRouter(s) managing it.

b) The objects residing on the node to be upgraded are 'routed' to alternate nodes. (similar to a failure scenario).

c) The hardware is replaced, modified, or upgraded through whatever procedure is required (ie: opening the case and replacing memory, drives, etc).

d) The hardware is restored and brought back on line.

e) The hardware undergoes the 'melding' procedure to (re)join the 'collective' of nodes.

The following paragraphs describe how hardware evolution is related to software evolution:

It is possible in a 'dependency hierarchy' that upgrading the software for a given node will also require a hardware upgrade (it is anticipated that the ObjectRouter will have detailed knowledge of each node's hardware capabilities).

For instance, a node may require additional memory or a new audio card to support the latest version of Java upon which the application software is based. The disclosed system's dependency tree goes deeper than "software object versions" and extends into language versions, runtime library versions, and Operating System requirements as well. Operating System features may in turn rely on hardware availability. This means that at deployment time, a piece of software may notify the installer that the hardware required to support it is unavailable. This would then lead into the hardware upgrade (mentioned above) required to support the software transition.

IV.G. Classloader changes:

Classloaders are a mechanism utilized within some object-oriented languages for loading object classes into memory. A common technique for changing the behavior of class loading within such systems is sometimes referred to as "classloader chaining." Utilizing this technique, varying behaviors can be given to

various classes loaded within the system. An example of classloader chaining is shown in FIG. 7. For instance, a "Primordial Classloader" 7.1 could be used to load language specific objects and grant these objects maximum permission to access the host system's files, memory, and network. Sub-classloaders which are 'chained' to the parent (primordial) classloader could behave differently. For instance, The "BootTime ClassLoader" 7.2 could add the behavior of loading classes stored in a remote server. The ClassLoader associated with Application Objects 7.4 could place security restrictions on those objects, prohibiting them from accessing the local machine's files. One particular limitation of a standard classloader 'chain' is that it provides no mechanism for handling multiple versions of a single object class.

The system described herein has solved this problem by creating a ClassLoader Hierarchy, rather than a chain structure. FIG.8 shows an example of a classloader hierarchy with multiple object versions loaded. Utilizing a classloader hierarchy permits a host node in the disclosed system to load multiple versions of the same object class simultaneously.

This, when combined with other disclosed techniques permits a seamless transition between software versions. Using traditional classloader techniques, one would need to terminate the software process and restart it in order to load a new version of a class that had already been loaded into memory.

IV.G.1. Example:

A User or the System starts a software process and loads class "Truck", version 1:

```
class Truck {
        static  String versionID = 1;
        public void startEngine() {
        }
        public void moveFaster() {
        }
}
```

Later, the definition of class "Truck" is changed and version 2 is produced, as shown below:

```
 class Truck {
        static  String versionID = 2;
        public void warmGlowPlugs() {
        }
        public void startEngine() {
        }
        public void accelerate() {
        }
 }
```

Normally, a restart of the process would be required in order to load version 2 of Truck. Additionally, loading version 2 of "Truck" would remove version 1 of class "Truck" from memory and break any software relying on usage of the moveFaster() method in version 1. Utilizing the specialized classloader hierarchy design, both version 1 and version 2 can co-exist within the process without any interruption of service to clients or other objects relying on either version 1 or version 2.

IV.H. JVM Extensions:

Several extensions are provided to the Java Virtual Machines (JVM) as follows:

- **Extended Process Control:** APIs are provided for tracking process resource utilization, starting, changing the priority of, and stopping processes.
- **Extended Thread Control:** APIs are provided for obtaining greater control over executing system threads.
- **Virtual Processes within VM:** Pseudo processes within the VM are created through a combination of threads/thread groups and specialized classloader mechanisms.
- **Specialized Classloader with support for versioning:** Support is provided for loading multiple versions of the same object simultaneously within the same VM.

It should be noted that not all the above extensions are strictly required, but they provide for better system operation.

IV.I. Failure of object router with multiple instances of object routers:

The disclosed technique provides for recovery from failure of object routers. If there are two or more nodes functioning as object routers, each object router will be paired with at least one secondary mirror. Each object router is aware (directly or indirectly through a hierarchy) of other object routers within the system. A failure is detected in a fashion similar to what is described in section IV.I. Upon failure the following steps are performed.

1) Nodes utilize the services of secondary object routers (which become primary).
2) The 'replicated' object router is 'cloned' (using techniques similar to the object replication described in section IV. E. above) and activated on at least one secondary node.
3) All nodes managed by the object router are notified of the new secondary location(s).
4) Other object routers are notified of the new secondary location(s).

In addition to the above steps, the following steps are optionally performed:

1. System Administrator is notified of failure.
2. Other systems are notified of failure.
3. Failed object router is requested to reboot into a 'recovery' mode.

IV.J. Failure of object router when only one instance of object router exists:

Upon failure (detected as described in section IV.I), the following steps are performed:

1) Remaining nodes 'elect' a new node to function as object router (if only one node exists, that node will perform both application processing and router functions)
2) Node elected to function as object router starts object router module and reads last known state of prior router from a database, set of flat files, or other data server. [if possible]
3) Announcement is broadcast to nodes notifying them of the availability of new object router.
4) Newly started object router requests from each node it directly manages:
   a) Current status including system capabilities and current 'load/utilization' statistics.
   b) List of current objects residing on the managed node.

In addition to the above steps, the following steps are optionally performed:

System Administrator is notified of failure.

Other systems are notified of failure.

Failed object router is requested to reboot into a 'recovery' mode.

IV.K. Failure of Node

When a node within the disclosed system fails, that failure can be detected through several means including but not limited to:

1) A 'proxy' object from another node is unable to reach the node. The failure is reported to the object router.
2) A node fails to respond to 'heartbeat' requests from the object router.
3) A node fails to report either status or statistics to the object router within a given time span or upon request.

When any node failures are detected, the object router responds by:

1) Notifying other object routers and managed nodes that the node has failed.
2) For each object that resided on the failed node:
   - the replicated objects (located on other nodes) become live/primary.
   - Object router locates additional computing resources within the system.
   - Object router locates the remaining object 'replicates' and makes at least one additional "clone" / replicate on secondary nodes.
   - notifies other object routers and nodes of the new location of the replicates.
3) Notifies the system administrator that the node has failed and is now 'offline'
4) Optionally notifies other systems that the node has failed.

5) If possible, reboots the system node into 'recovery' mode – which attempts to perform self-diagnostics and repair. (Reinstall/checksums of installed software and hardware, and other related diagnostic tasks are performed).

It is assumed that every object within the system will normally have at least one replicate at all times. In higher availability configurations, objects may have multiple replicates located on multiple nodes, increasing the fault tolerance level. "Standard" configurations (with at least one replicate) will tolerate the failure of any one node within a small time delta. "Highly-redundant" configurations (with multiple replicates) may tolerate multiple node failures simultaneously.

IV.L. Encryption

Within the disclosed system, all messages and network traffic sent between nodes may be encrypted utilizing techniques including but not limited to: public and private key encryption, digital certificates, digital signatures, secure message digests, and shared private key encryption of data.

Additionally, the encryption algorithms and approaches utilized may include but are not limited to: DES, triple DES, BlowFish, RSA public key, SSL, TLS, PKCS, IMAP, LDAP, Kerberos, S/MIME, RC4, Diffie-Hellman, and DSA public key.

Various authentication schemes may be utilized. This involves a variety of digital signature and biometric approaches including but not limited to: PGP, LDAP, Kerberos, thumbprint scanners, handprint scanners, retinal scanners, voiceprint recognition, passwords, one-time passwords, hardware based key generators,

smart cards, embedded devices, or any combination thereof. Such authentication schemes may be applied at the system, user, transport, object, or binary level.

Similar encryption and authentication techniques may also be applied to all messages, checksums, and network traffic sent between components of the system including but not limited to melders, object routers, migrators, replicators, code version servers, classloaders, node controllers, and node resource monitors.

Furthermore, within the disclosed system, messages and network traffic sent to user applications, end-users, and external systems may also be encrypted and authenticated in a similar fashion.

Those skilled in the art of encryption may recognize other hardware and software techniques which could also be easily applied to the disclosed system.

IV.M. End-User Benefits:

The end-users of a computing platform embodying the disclosed techniques and teachings experience a variety of benefits. FIG.9 shows examples of locations where a system embodying the disclosed techniques can be deployed and utilized..

For the purpose of this disclosure, a Client System is described as a machine, computer, device, or software presenting a user interface to an end user of computing services.

A Client-Server system that has a client system that interacts with a server system utilizing the disclosed teachings and techniques, and hosting customer's

services experiences these benefits. The Client System will not experience performance degradation or failure due to hardware/software upgrades or failures within the Server System.

Similarly, a Server-Server system with a server system 'A' interacting with a server system 'B' utilizing the disclosed teachings and techniques, and hosting customer's services experiences these benefits. The Server System 'A' will not experience performance degradation or failure due to hardware/software upgrades or machine failures within the Server System 'B'

IV.N. End-User Experience:

The flexibility of the preferred embodiment embodying the disclosed teachings and techniques permits it to be deployed in any system that requires growth, stability, and optimal resource management. From an end-user's point of view, the system meets the same standard of dependability that basic utilities such as water, electricity, and phone provide.

When software application users (or "Web surfers") encounter messages like "Server cannot be located...," "Host did not respond...," "System unavailable...," or timeouts including no response from the server, they are annoyed and try again for perhaps a second time before giving up. Consumers of today's networked systems assume that better technology is being constructed but currently available systems are not yet ready to provide these essential services . Without solving chronic problems hidden in the computing environments that provide the backbone

services users depend on, these users' hopeful progress will instead turn into more painful experiences.

Everyday, we watch TV, listen to radio, read papers, and talk on the phone. These are some of the "essential" activities that we take for granted and demand be available to us 24 hours a day 7 days a week. In addition to the traditional necessities (such as telephone and electricity), we increasingly look toward ways to use continuously evolving information technology to bring new services for better productivity and entertainment. Services like on-demand video and audio over the internet will soon be widely accepted and these services will be expected to perform both continuously and reliably. An individual watching the SuperBowl broadcast over the Internet would expect perfect delivery of the event (including commercials). A system that embodies the disclosed techniques guarantees this through its multi-layer redundancies. As more and more people subscribe to events broadcasted over the Internet (say from 1,000 to 1,000,000), such a system scales without ever needing to disrupt the services. Existing and new users all enjoy the service as if they are the first and only user. When a new feature such as a 3D broadcast is made available, users relying on the disclosed system are given the choice to add this feature without experiencing degradation of current services or needing to wait for system expansion. More familiar examples include the outages and flux of services that have occurred at major corporations including Ebay, Hotmail, Yahoo, and AOL as systems have strained to accommodate rapid growth and change. A system using the disclosed techniques, provides the end user with a true 24 hour X 365 day environment. Most production systems today have a "schedule" of downtimes for planned software and hardware maintenance.

Even during hardware and software upgrades a system embodying the disclosed techniques and teachings remains available for all users. End users can continue interacting with the system ignorant of the fact that a software or hardware upgrade is underway. This provides unprecedented levels of availability. An important feature of the system embodying the disclosed technique is that the users will not experience the downtime typically associated with machine failures, replacements, and upgrades.

IV.O. Customer Benefits:

A Customer of a system that uses the disclosed techniques is a direct purchaser of the system who utilizes it to provide reliable services to end-users. Likely customers include Fortune 500 Corporations, e-commerce/web companies, System Integrators, and Application Service Providers (ASPs).

Customers experience these benefits:

1) Reliability, Scalability, Availability: The system is by its nature highly redundant / fault tolerant and scales easily to meet increasing system demand.

2) Ease of Use: The system simplifies administration of large numbers of machines. Once a node in the system is loaded with necessary components, that system is administered by itself and the object router. The entire system can be administered from a single unified interface.

3) Incremental Capacity Upgrades: The system permits easy upgradability by "plugging in" (melding) additional nodes.

4) Failure Recovery: the system "routes around" node failures and self-heals (given available hardware resources).

5) Seamless software and hardware upgrades: The system permits software and hardware upgrades to occur while remaining 100% available to existing users.

IV.P. Process Of Evolution

The process of evolution is described herein from an end-user's perspective. A user actively utilizing an application built with Evolution will not notice any change in the system behavior while the system is evolving. Upon completion of an upgrade, the user will see the new features/functionality available to them.

IV.P.1. Example 1

Users of a video-on-demand system are selecting and watching videos for free on a new web site: FreeVid.com

FreeVid.com realizes they are losing money and need to begin charging for their services. While users continue to select and watch videos, the system is evolved and new software is deployed that will require them to enter a credit card number. After the system has completed it's upgrade, users are presented with a

screen informing them of the bad news, providing them with credits for 3 free viewings, and requesting payment for continued service.

IV.P.2. Example 2:

Users of a Bill Payment system that accesses their checking accounts are logged in and utilizing it's services. While they remain online, the system is upgraded to also accept credit card payments.

IV.Q. From a Customer's perspective

A customer purchasing a system embodying the disclosed techniques perform the tasks described herein. When the customer is ready to upgrade their system, they log into a system administration console. From the administration console, they will select a software module or set of modules to upgrade/replace. They then initiate the upgrade by pressing a 'start' or 'begin' button. A progress meter will show the upgrade occurring, and upon completion the new feature set will be available. The administrator may then 'turn off' or discard prior software modules that are no longer needed.

IV.R. From the Developer's perspective:

A customer building software to be deployed on a system embodying the disclosed techniques performs the tasks described herein. The developer will construct software modules that comply with the APIs for the disclosed system. They will then compile their software. After compilation, either a pre-processor, language compiler or "post-processor" will analyze the class/object files and insert versioning information. It will then perform an object class level 'diff' between software versions. When possible the system will create automatic transform functions. In other cases, the developer may need to select from pre-defined or create custom "transform" functions that enable the objects to migrate from one version to the next. Finally, the modified class/object files and transform functions are packaged into an archive format for delivery and deployment.

IV.S. Computer program product

Apart from systems and methods, computer program products are also within the scope of the disclosed teaching. These computer program products comprise instructions on a computer readable medium that enable a computer to perform the methods disclosed herein. The instructions are not limited, and include but not limited to, source code, object code and executables. The computers on which the instructions are implemented include, but not limited to, minis, micros, and mainframes. The computer readable medium includes, but not limited to, floppies, RAMs, ROMs, hard drives, magnetic tapes, cartridges, CDs, DVDs, and internet downloads.

It should be noted that the system components and the processes disclosed can be implemented in any way in a selected network system. Though a software implementation is preferable, it can also be implemented by a hardware or a hardware/software combination. For example, a subset of components or processes can be implemented using pure hardware as by using ASICs. Similarly, in the software implementation, there is no restriction regarding the choice or level of computer languages. For example, higher level languages like Java, C++, etc, could be used or lower level languages including assembly and machine languages could be used. Or else, a combination of computer languages could be used. Though some of the preferred embodiments described herein might appear to reveal a Java-based implementation, this is not meant to be restrictive. The choice of language and implementation would depend on many factors including the needs of the user and availability of resources.

Also, compatibility issues should not be treated as being restrictive. For, example, it should be noted that a system embodying the disclosed teachings and techniques might be loaded on existing network systems. This might cause some compatibility issues. However, these compatibility issues can be overcome by a skilled practitioner without deviating from the scope of the disclosed teachings.

Other modifications and variations to the invention will be apparent to those skilled in the art from the foregoing disclosure and teachings. Thus, while only certain embodiments of the invention have been specifically described herein, it will be apparent that numerous modifications may be made thereto without departing from the spirit and scope of the disclosed teachings and techniques.

WHAT IS CLAIMED IS:

1. A method for creating a scalable, fault tolerant computing platform on a network, said platform comprising a plurality of nodes, said method comprising:

a) initiating melding during initialization, wherein during said melding said plurality of nodes are treated as a collection and each node from said plurality of nodes joins said collection, and during melding at least a first object router is assigned;

b) migrating objects between said plurality of nodes; and

c) replicating objects across said plurality of nodes.

2. The method of claim 1, wherein during melding each node joins the collection using a process comprising:

(a)(i) the node finding a nearby object router if the object router can be found, otherwise becoming the object router; and

(a)(ii) the node reporting available resources to the object router.

3. The method of claim 1, wherein during melding each node joins the collection using a process further comprising:

(a)(iii) assigning network address for the node; and

(a)(iv) updating components of the node.

4. The method of claim 2, wherein the available resources comprise memory, at least one processor and drive storage space.

5. The method of claim 2, wherein the available resource comprises network bandwidth.

6. The method of claim 2, wherein the available resources comprise drive storage space.

7. The method of claim 2, wherein said updating is done using data stored in a code version server.

8. The method of claim 2, wherein it is ensured that at least one mirror of an object router exists all the time.

9. The method of claim 1, wherein the object router manages resources and distributes load evenly across the plurality of nodes.

10. The method of claim 1 wherein the object router maintains knowledge of locations of at least a subset of all nodes in the computing platform.

11. The method of claim 1 wherein the object router relocates or replicates objects within the plurality of nodes.

12. The method of claim 1, wherein said migration of an object is performed using a process comprising:

(b)(i) locking the object;

(b)(ii) converting the object to a byte stream at a first node in the plurality of nodes;

(b)(iii) transferring the byte stream across the network;

(b)(iv) converting the byte stream back to the object at a second node in the plurality of nodes;

(b)(v) updating a location of the object in the object router; and

(b)(vi) unlocking the object.

13. The method of claim 1, wherein said replication of an object is performed by a process comprising:

(c)(i) locking the object at a first node;

(c)(ii) forming a duplicate object, ;

(c)(iii) locating the duplicate object at a second node; and

(c)(iv) unlocking the object.

14. The method of claim 13, further comprising:

(c)(v) subscribing to events associated with the object.

15. The method of claim 13, wherein said duplication is performed by creating a byte stream from the object and converting said byte stream into a duplicate object.

16. The method of claim 1, wherein the computing platform allows for evolution, wherein said evolution provides for transitioning of system software and hardware such that a seamless upgrade is possible.

17. A method for performing evolution, said evolution being performed to enable a seamless transitioning of software, said method comprising:

(a) evaluating a software or hardware to be installed or upgraded;

(b) creating a dependency hierarchy tree of all objects in the computing platform at compile time;

(c) performing object class diff comparison between old and new versions of at least one class of objects;

(d) creating a transform function for each state variable that is changed;

(e) selecting an object that is an instance of the selected object class;

(f) locking the selected object;

(g) creating uninitialized instance of the new version of the selected object;

(h) initializing variables of the new object class instance by applying appropriate transform functions to the old object version;

(i) unlocking the selected object; and

(j) repeating steps e-i for each instance of the object class.

18. The method of claim 17, further comprising:

(l) repeating steps e-k for each object class in the computing platform.

19. The method of claim 17, further comprising:

(m) deploying client-side software.

20. The method of claim 17, further comprising:

(n) destroying older version of the objects if multiple versions of the objects are not required.

21. The method of claim 1, wherein the object router maintains multiple instances of each of said objects.

22. The method of claim 1, wherein a failure of a first node from the plurality of nodes is detected when a proxy object from a second node is unable to reach the first node.

23. The method of claim 1, wherein a failure of a node from the plurality of nodes is detected when the node fails to respond to heartbeat requests from the object router.

24. The method of claim 1, wherein a failure of node from the plurality of nodes is detected when the node fails to report either status or statistics to the object router.

25. The method of claim 1, wherein when a node from the plurality of nodes fails, said node being called a failed node, said at least one object router responds using a process comprising:

(i) notifying other object routers other than said at least one object router;

(ii) selecting an object that resided in the failed node;

(iii) making replicated object corresponding to the selected object residing in nodes other than the failed node primary;

(iv) locating additional computing resources within the computing platform;

(v) creating new replicates for the newly created primary objects in step iii and locating them in nodes other than nodes in which they originally resided; and

(vi) notifying said other object routers of new location of the new replicates.

26. The method of claim 25, further comprising:

(vii) notifying a system administrator about the failed node being off-line.

27. The method of claim 25, further comprising repeating steps ii-vi for all objects in the failed node.

28. The method of claim 1, wherein when an object router from said at least one object router fails, said object router that fails being called a failed object router, the computing platform responds using a process comprising:

(i) utilizing services of object routers other than said failed object router;

(ii) replicating the failed object router to create a replicate object router;

(iii) locating the replicate object router on at least one functional node ; and

(iv) notifying at least a subset of the nodes of a new location of the replicate object router.

29. The method of claim 28, further comprising:

(v) notifying at least a subset of other object routers of the new location of the replicate object router.

30. The method of claim 1, when said object router is aware of at least a subset of other object routers.

31. A network computing platform comprising a plurality of nodes, said system comprising:

at least an object router ,

wherein said object router is capable of managing resources and distributing load evenly across the plurality of nodes, and

wherein said object router maintains a knowledge of locations of at least a subset of all nodes in the computing platform.

32. The platform of claim 31 wherein the object router relocates objects within the plurality of nodes.

33. The platform of claim 31, wherein the object router maintains multiple instances of each object that is deployed in the platform.

34. The platform of claim 31 wherein each of said nodes comprise:

an object melder that treats the plurality of nodes as a collection that performs melding to create said plurality of nodes and said at least one object router.

35. The platform of claim 34, further comprising an object versioner that maintains multiple versions of all objects within the platforms.

36. The platform of claim 34, further comprising an object replicator that replicates an object to maintain a duplicate of the object in a node different from the node in which the object resided.

37. The platform of claim 34, further comprising an object migrator that enables an object to migrate from one node to another.

38. The platform of claim 34, further comprising a resource monitor.

39. The platform of claim 34, further comprising a resource controller.

40. The platform of claim 34 further comprising an application object.

41. The platform of claim 34 wherein each of said nodes further comprise:

a class loader that is responsible for loading classes of objects, wherein the class loader permits multiple versions of a same class to co-exist.

42. The platform of claim 31 further comprising:

an evolution module, wherein the evolution module allows the computing platform to perform evolution, wherein said evolution provides for transitioning of system software and hardware such that a seamless upgrade is possible.

43. The platform of claim 31, wherein a failure of a first node from the plurality of nodes is detected by the platform when a proxy object from a second node is unable to reach the first node.

44. The platform of claim 31, wherein a failure of a node from the plurality of nodes is detected by the platform when the node fails to respond to heartbeat requests from the object router.

45. The platform of claim 31, wherein a failure of node from the plurality of nodes is detected by the platform when the node fails to report status or statistics to the object router.

46. The platform of claim 31, wherein when a node from the plurality of nodes fails, said node being called a failed node, said at least one object router is capable of notifying other object routers other than said at least one object router.

47. The platform of claim 31, wherein when a node from the plurality of nodes fails, said node being called a failed node, said at least one object router is capable of notifying other nodes

48. The platform of claim 31, wherein when a node from the plurality of nodes fails, said node being called a failed node, said at least one object router is capable of making replicated object corresponding to at least a subset of objects residing in nodes other than the failed node primary.

49. The platform of claim 31, when a node from the plurality of nodes fails, said node being called a failed node, said at least one object router is capable of locating additional computing resources within the computing platform and creating new replicates for newly created primary objects and locating them in nodes other than nodes in which they originally resided.

50. The platform of claim 49, wherein the platform is capable of notifying said other object routers of new location of the new replicates.

51. The platform of claim 49, wherein the platform is capable of notifying a system administrator about the failed node being off-line.

52. The platform of claim 31, wherein when an object router from said at least one object router fails, the computing platform is capable of utilizing services of object routers other than said failed object router.

53. The platform of claim 31, wherein when an object router from said at least one object router fails ,the computing platform is capable of replicating the failed object router to create a replicate object router and locating the replicate object router on at least one functional node.

54. The platform of claim 53, wherein the platform is further capable of notifying at least a subset of the nodes of a new location of the replicate object router.

55. The platform of claim 31, wherein said object router is aware of at least a subset of other object routers.

56. An object router for a network computing platform comprising a plurality of nodes, wherein said object router is capable of managing resources and distributing load evenly across the plurality of nodes, and wherein said object router maintain a knowledge of locations of at least a subset of all nodes in the computing platform.

57. The object router of claim 56, wherein the object router relocates objects within the plurality of nodes.

58. The object router of claim 56, wherein the object router maintains multiple instances of each object that is deployed in the platform.

59. The object router of claim 56, wherein when a node from the plurality of nodes fails, the object router is capable of notifying other object routers other than said object router.

60. The object router of claim 56, wherein when a node from the plurality of nodes fails, the object router is capable of making replicated object corresponding to at least a subset of objects residing in nodes other than the failed node primary.

61. The object router of claim 56, wherein when a node from the plurality of nodes fails, the object router is capable of locating additional computing resources within the computing platform and creating new replicates for the new primary objects and locating them in nodes other than nodes in which they originally resided.

62. The object router of claim 56, wherein the object router is aware of at least a subset of other object routers.

63. A computer program product including a computer-readable medium comprising instructions to enable a computer to implement a method for creating a scalable, fault tolerant computing platform on a network, said platform comprising a plurality of nodes, said instructions comprising instructions for:

a) initiating melding during initialization, wherein during said melding said plurality of nodes are treated as a collection and each node from said plurality of nodes joins said collection, and during melding at least a first object router is assigned;

b) migrating objects between said plurality of nodes; and

c) replicating objects across said plurality of nodes.

64. The program product of claim 63, wherein during melding each node joins the collection using a process comprising:

(a)(i) the node finding a nearby object router if the object router can be found, otherwise becoming the object router; and

(a)(ii) the node reporting available resources to the object router.

65. The program product of claim 63, wherein during melding each node joins the collection using a process further comprising:

(a)(iii) assigning network address for the node; and

(a)(iv) updating components of the node.

66. The program product of claim 64, wherein the available resources comprise memory, at least one processing unit and drive storage space.

67. The program product of claim 63, wherein the available resource comprises network bandwith.

68. The program product of claim 64, wherein said updating is done using data stored in a code version server;

69. The program product of claim 64, wherein it is ensured that at least one mirror of an object router exists all the time.

70. The program product of claim 63, wherein the object router manages resources and distributes load evenly across the plurality of nodes.

71. The program product of claim 63, wherein the object router maintains a knowledge of locations of at least a subset of all nodes in the computing platform.

72. The program product of claim 63 wherein the object router relocates objects within the plurality of nodes.

73. The program product of claim 63, wherein said migration of an object is performed using a process comprising:

(b)(i) locking the object;

(b)(ii) converting the object to a byte stream at a first node in the plurality of nodes;

(b)(iii) transferring the byte stream across the network;

(b)(iv) converting the byte stream back to the object at a second node in the plurality of nodes; and

(b)(v) updating the location of the object in the object router; and

(b)(vi) unlocking the object.

74. The program product of claim 63, wherein said replication of an object is performed by a process comprising:

(c)(i) locking the object at a first node;

(c)(ii) forming a duplicate object;

(c)(iii) locating the duplicate object at a second node;

(c)(iv) subscribing to events associated with the object; and

(c)(v) unlocking the object.

75. The program product of claim 74, wherein said duplication is performed by creating a byte stream from the object and converting said byte stream into a duplicate object.

76. The program product of claim 63, wherein the computing platform allows for evolution, wherein said evolution provides for transitioning of system software and hardware such that a seamless upgrade is possible.

77. A computer program product including a computer readable medium comprising instructions for enabling a computer to perform evolution, said evolution being performed to enable a seamless transitioning of software, said instruction including instructions for:

evaluating a software or hardware to be installed or upgraded;

(b) creating a dependency hierarchy tree of all objects in the computing platform at compile time;

(c) performing object class diff comparison between old and new versions of at least one class of objects;

(d) creating a transform function for each state variable that is changed;

(e) selecting an object that is an instance of the selected object class;

(f) locking the selected object;

(g) creating uninitialized instance of the new version of the selected object;

(h) initializing varialbles of the new object class instance by applying appropriate transform functions to the old object version;

(i) unlocking the selected object; and

(j) repeating steps e-i for each instance of the object class.

78. The program product of claim 77, further comprising:

(l) repeating steps d-j for each object class in the computing platform.

79. The program product of claim 77, further comprising:

(m) deploying client-side software.

80. The program product of claim 77, further comprising:

(n) destroying older version of the objects if multiple versions of the objects are not required.

81. The program product of claim 63, wherein the object router maintains multiple instances of each of said objects.

82. The program product of claim 63, wherein a failure of a first node from the plurality of nodes is detected when a proxy object from a second node is unable to reach the first node.

83. The program product of claim 63, wherein a failure of a node from the plurality of nodes is detected when the node fails to respond to heartbeat requests from the object router.

84. The program product of claim 63, wherein a failure of node from the plurality of nodes is detected when the node fails to report status or statistics to the object router.

85. The program product of claim 63, wherein when a node from the plurality of nodes fails, said node being called a failed node, said at least one object router responds using a process comprising:

(i) notifying other object routers other than said at least one object router;

(ii) selecting an object that resided in the failed node;

(iii) making replicated object corresponding to the selected object residing in nodes other than the failed node primary;

(iv) locating additional computing resources within the computing platform;

(v) creating new replicates for the newly created primary objects in step iii and locating them in nodes other than nodes in which they originally resided; and

(vi) notifying said other object routers of new location of the new replicates.

86. The program product of claim 85, further comprising:

(vii) notifying a system administrator about the failed node being off-line.

87. The program product of claim 85, further comprising repeating steps ii-vi for all objects in the failed node.

88. The program product of claim 63, wherein when an object router from said at least one object router fails, said object router that fails being called a failed object router, the computing platform responds using a process comprising:

i) utilizing services of object routers other than said failed object router;

ii) replicating the failed object router to create a replicate object router;

iii) locating the replicate object router on at least one functional node ; and

iv) notifying a subset of the nodes of a new location of the replicate object router.

89. The program product of claim 63, when said object router is aware of at least a subset of other object routers.

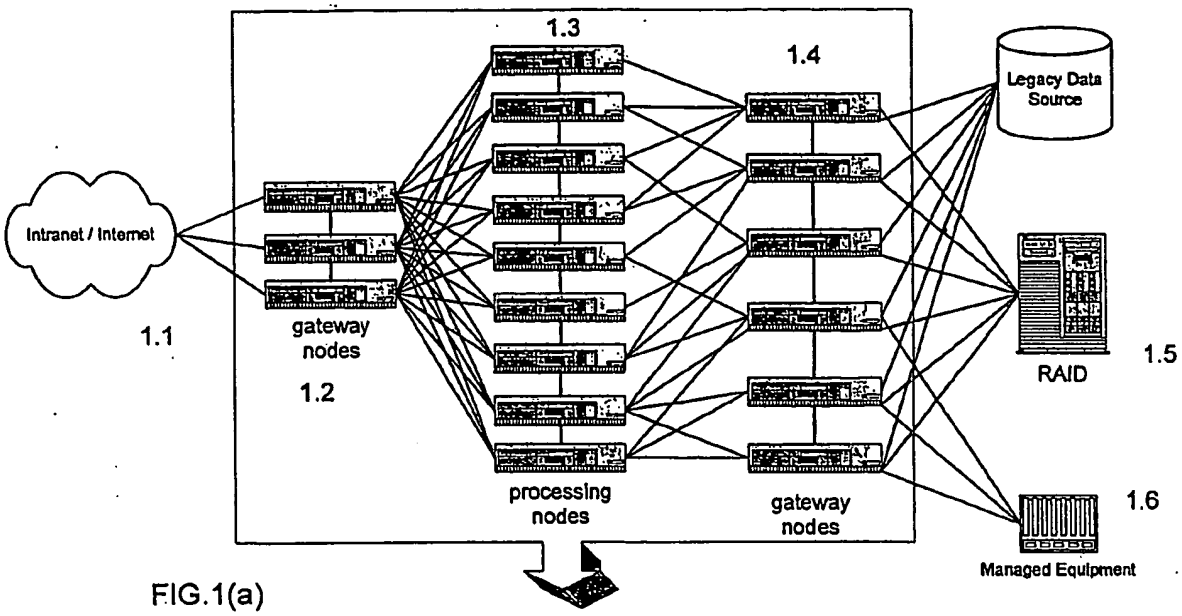


FIG.1(a)

Concept Diagram

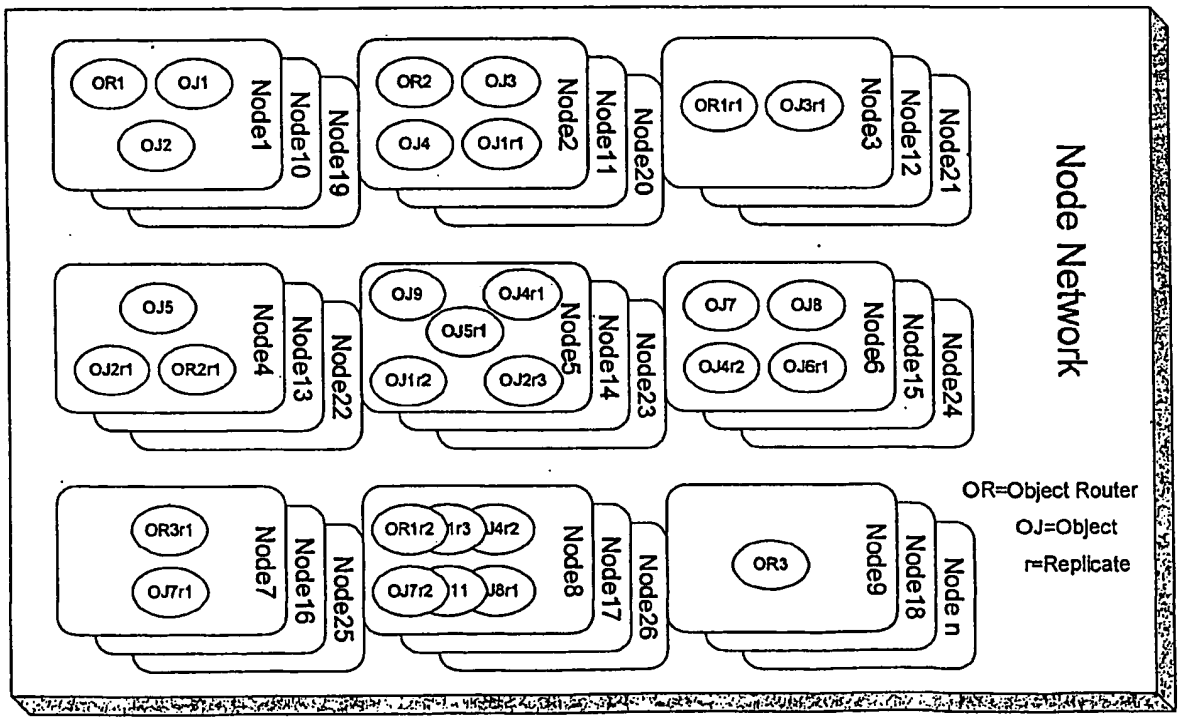


FIG.1(b)

Node Architecture

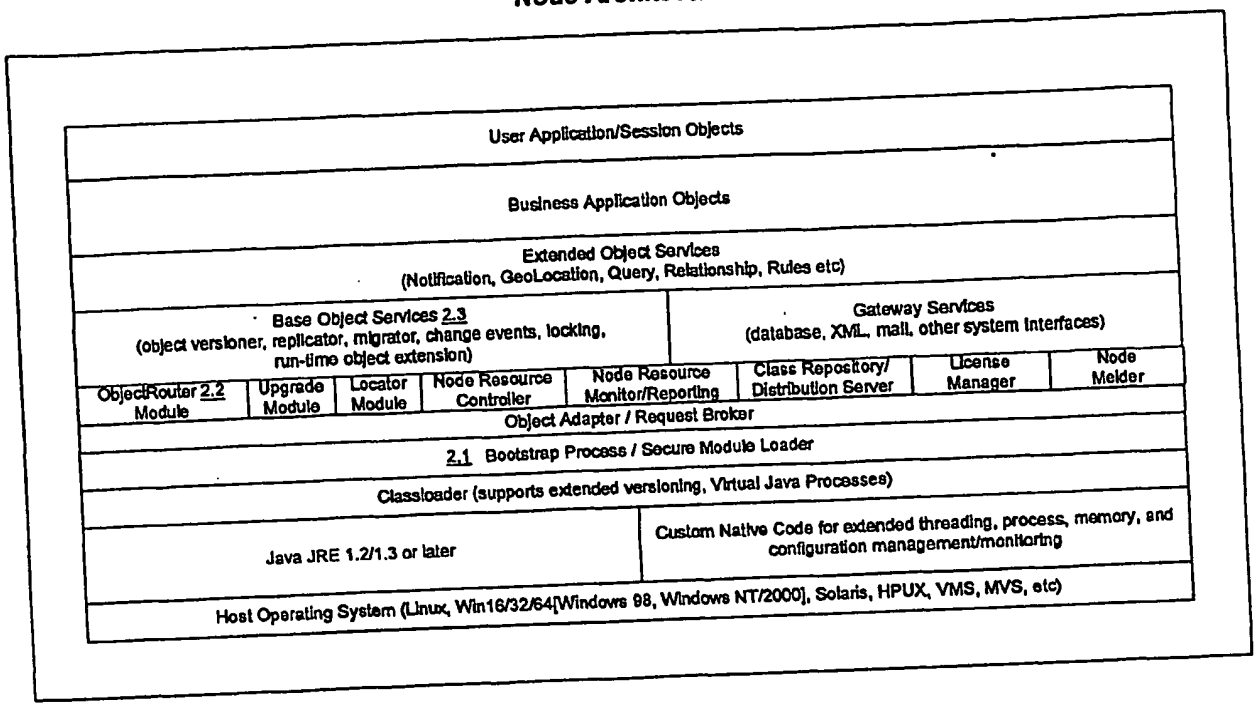


FIG.2

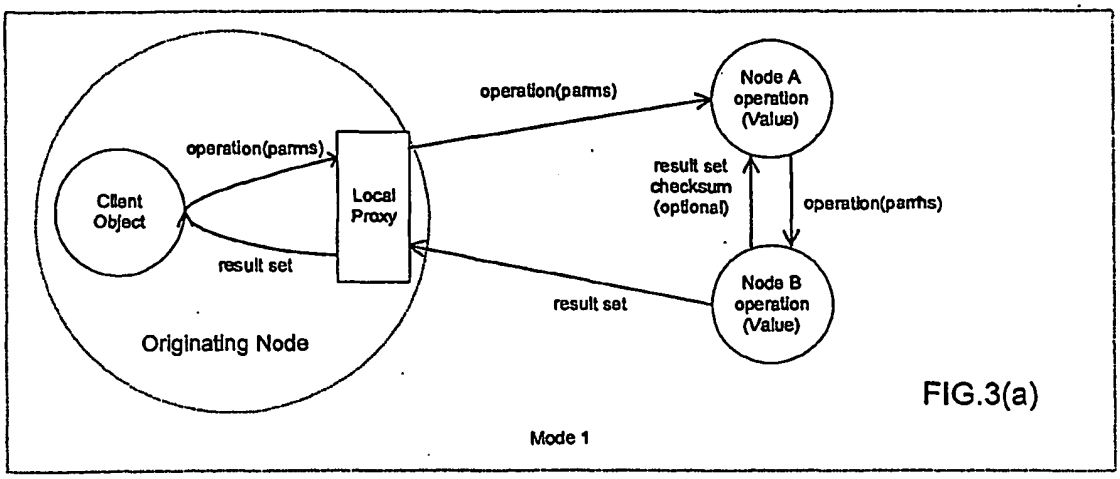


FIG.3(a)

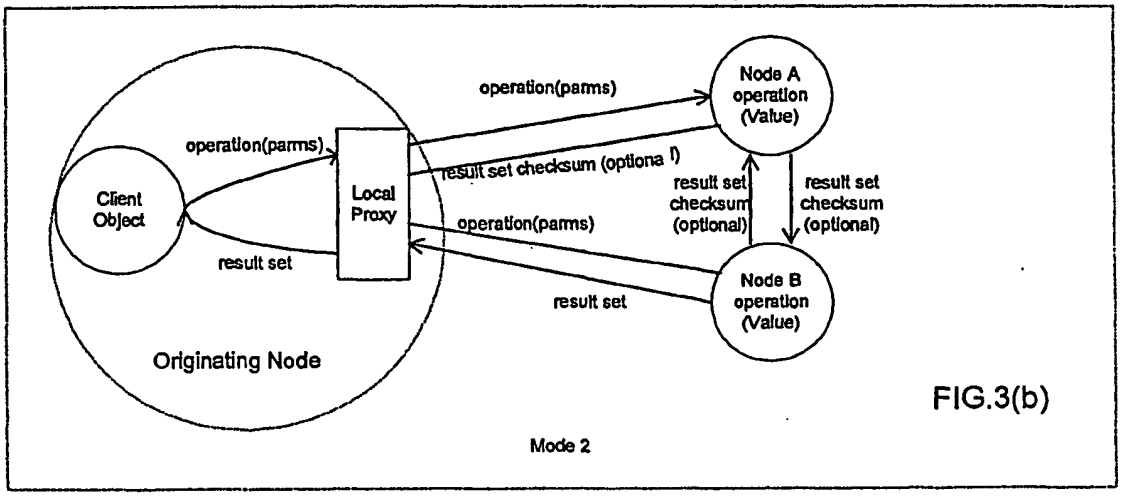


FIG.3(b)

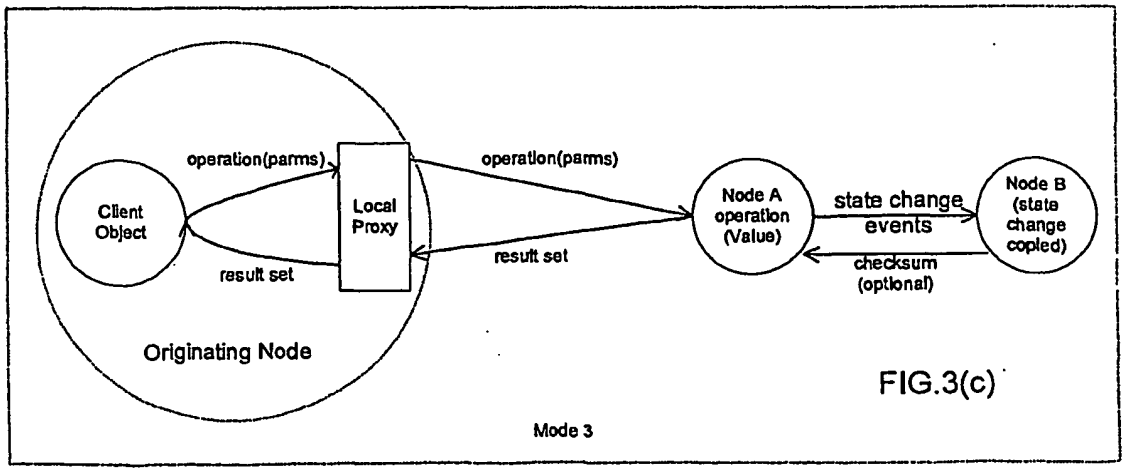
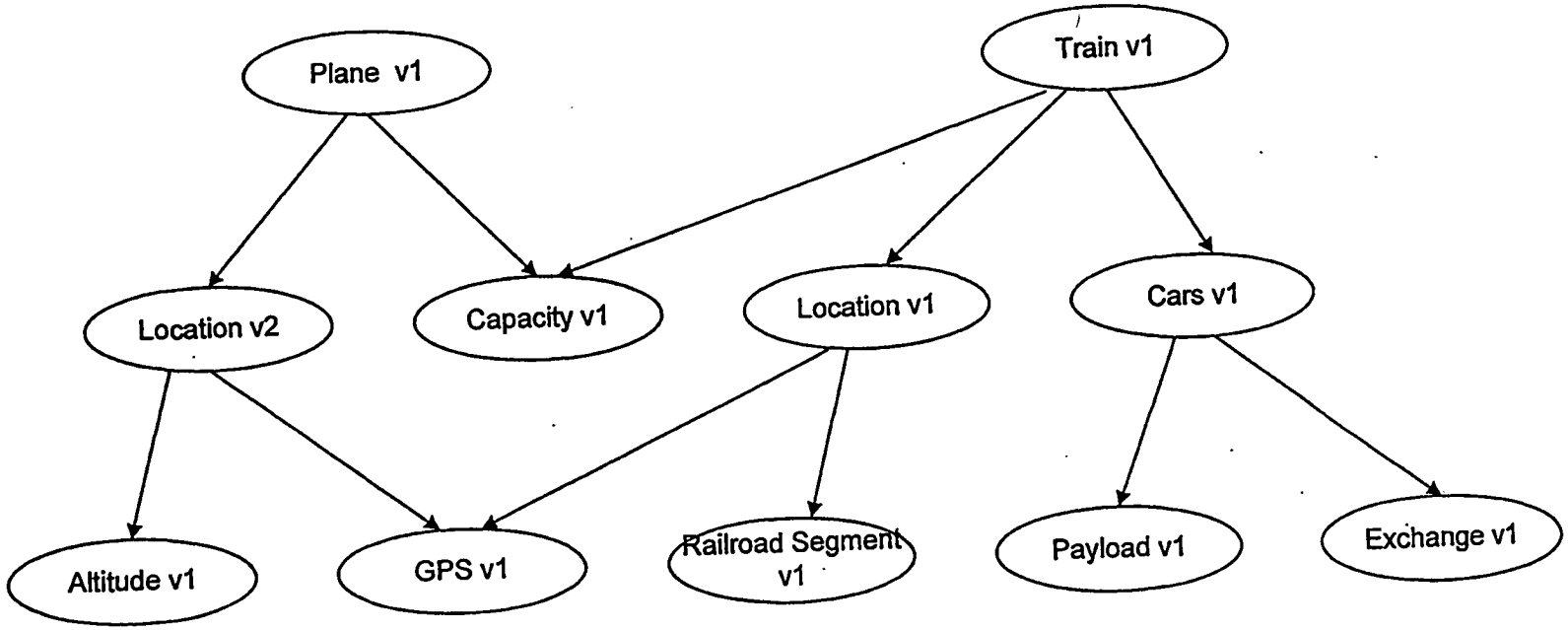


FIG.3(c)

Object Class Version Tree

FIG.4



Logical View

Embedded Object Versioning

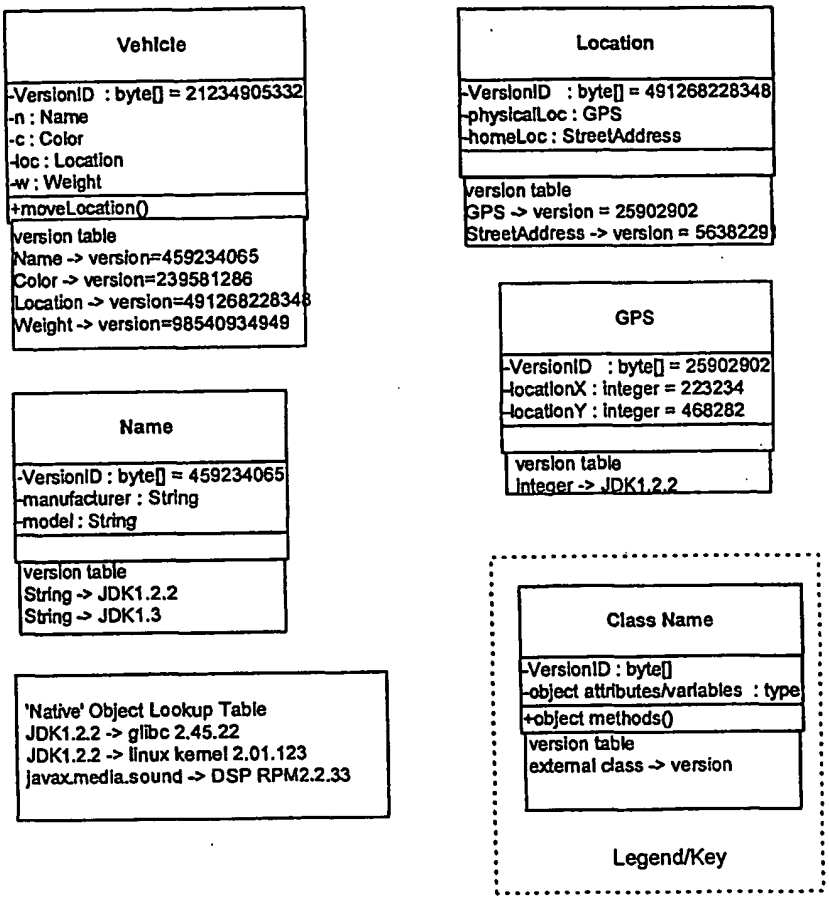


FIG.5(a)

Version Lookup Table
ClassName: External Class -> Version

Vehicle: Name -> version=459234065
Vehicle: Color -> version=239581286
Vehicle: Location -> version=491268228348
Vehicle: Weight -> version=98540934949
Location: GPS -> version = 25902902
Location: StreetAddress -> version = 56382291
Name: String -> JDK1.2.2
Name: String -> JDK1.3
GPS: integer -> JDK1.2.2

FIG.5(b)

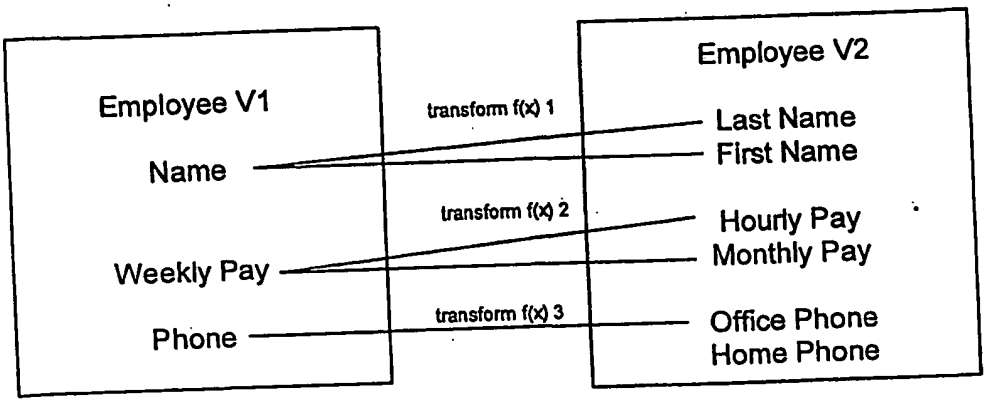

Employee V1
Name
Weekly Pay
Phone
transform f(x) 1
transform f(x) 2
transform f(x) 3
Employee V2
Last Name
First Name
Hourly Pay
Monthly Pay
Office Phone
Home Phone

FIG.6

Classloader Chaining

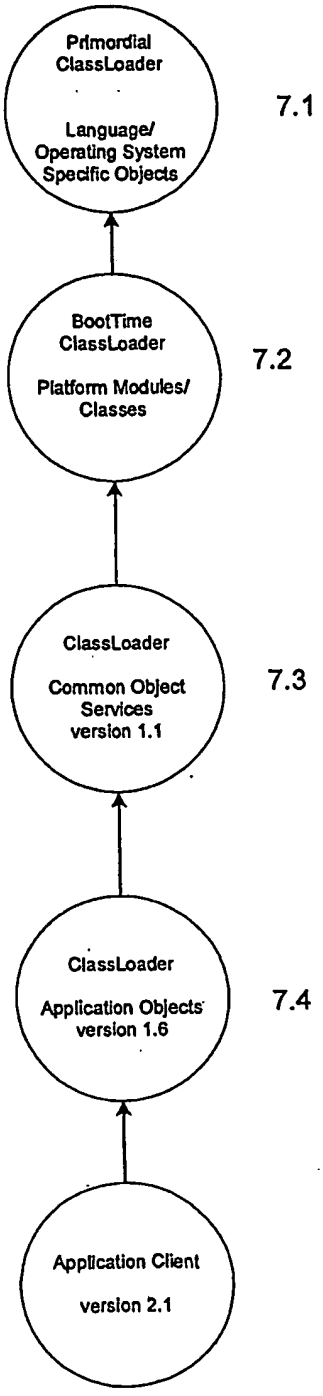


FIG.7

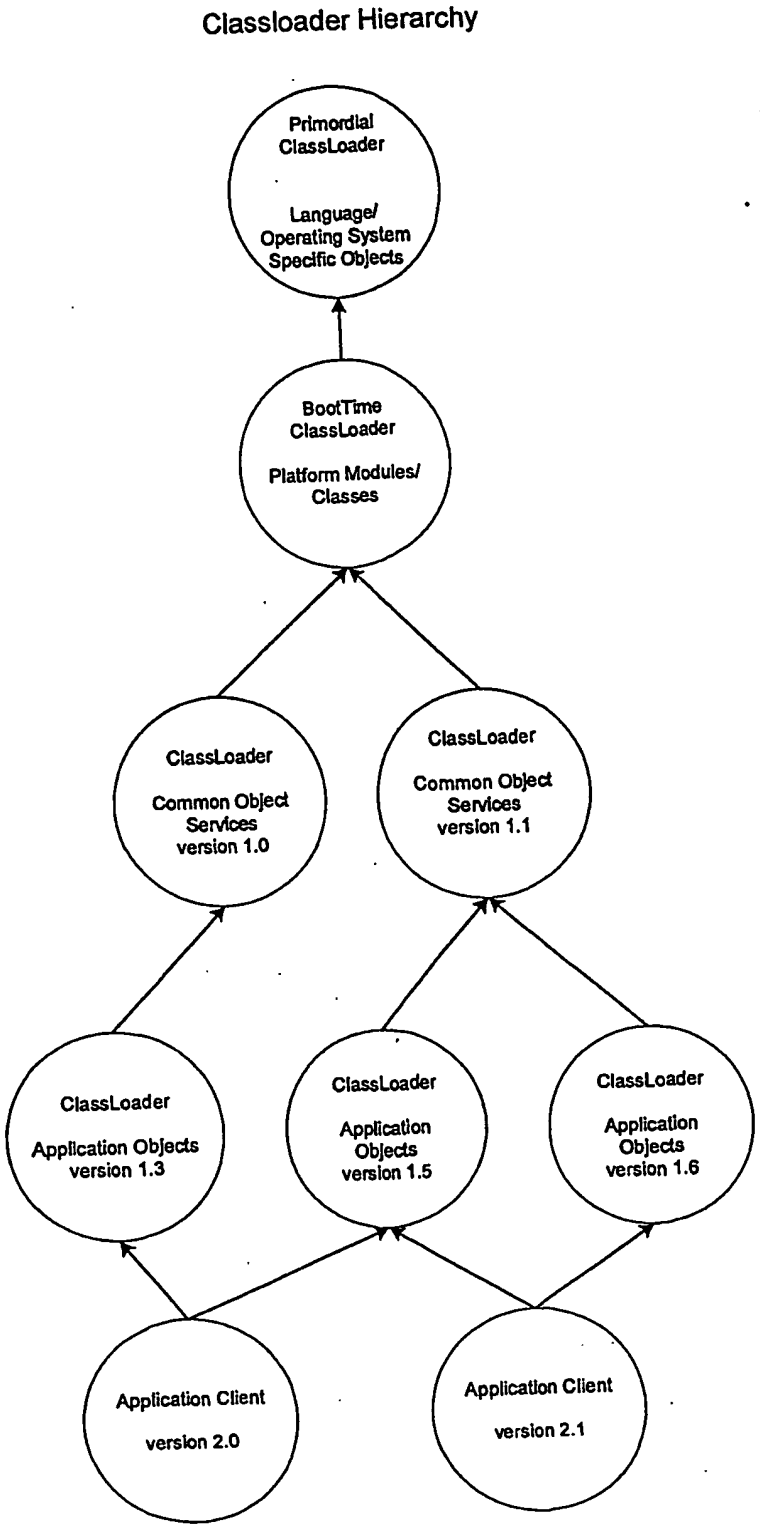

Classloader Hierarchy
Primordial ClassLoader
Language/ Operating System Specific Objects
BootTime ClassLoader
Platform Modules/ Classes
ClassLoader
Common Object Services version 1.0
ClassLoader
Common Object Services version 1.1
ClassLoader
Application Objects version 1.3
ClassLoader
Application Objects version 1.5
ClassLoader
Application Objects version 1.6
Application Client
version 2.0
Application Client
version 2.1

FIG.8

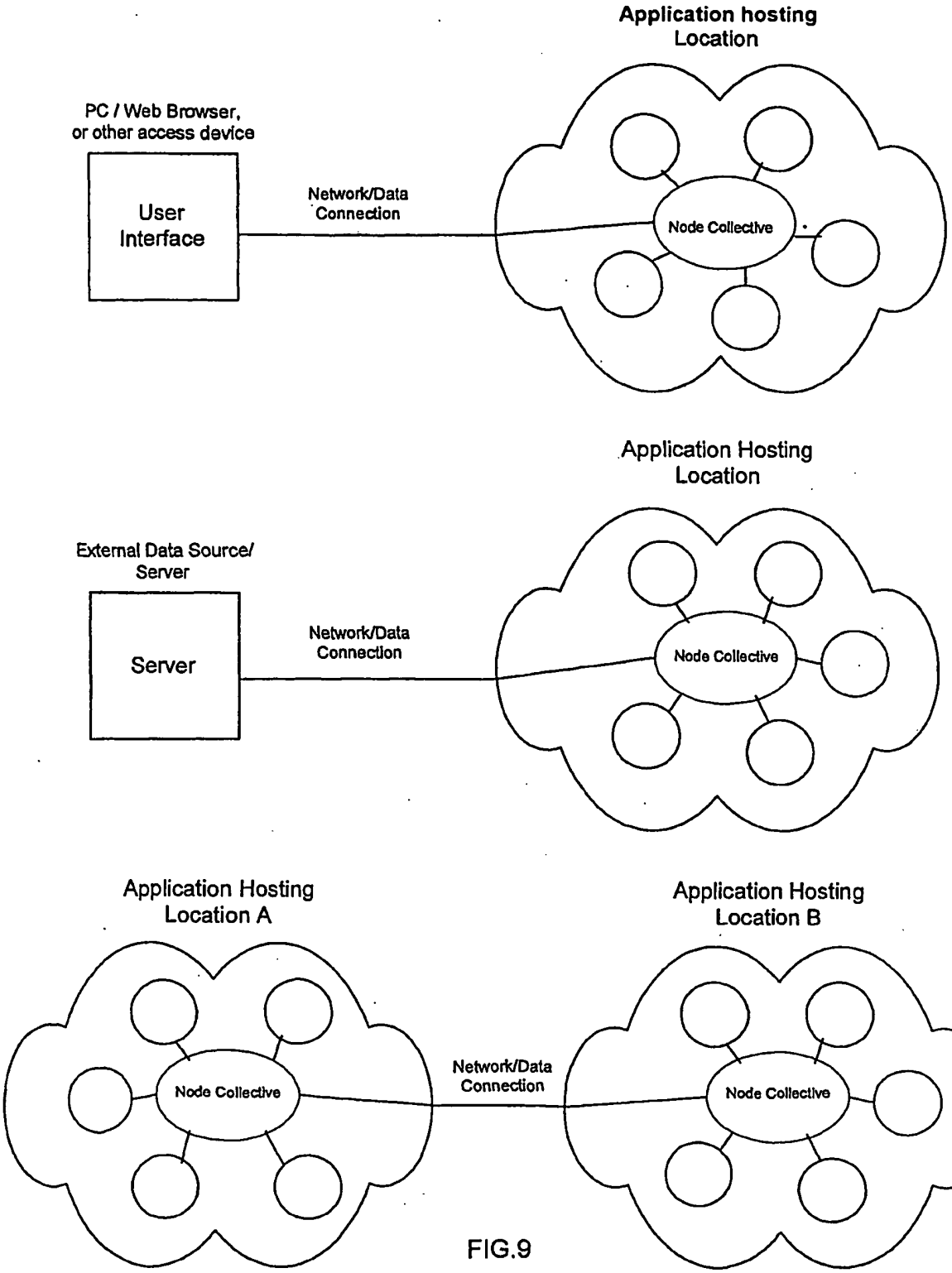

Application hosting Location
PC / Web Browser, or other access device
User Interface
Network/Data Connection
Node Collective
Application Hosting Location
External Data Source/ Server
Server
Network/Data Connection
Node Collective
Application Hosting Location A
Application Hosting Location B
Node Collective
Network/Data Connection
Node Collective

FIG.9